Extracting Business Rules from COBOL: A Model-Based Framework

Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, Jacques Perronnet

Extracting Business Rules from COBOL: A Model-Based Framework

Valerio Cosentino
AtlanMod, INRIA, EMN, Nantes, France
and IBM France
valerio.cosentino@fr.ibm.com

Jordi Cabot
AtlanMod, INRIA, EMN, Nantes, France
jordi.cabot@mines-nantes.fr

Patrick Albert
Philippe Bauquel
Jacques Perronnet
IBM France
albertpa, bauquel.p
jacques_perronnet@fr.ibm.com

***Abstract*—Organizations rely on the logic embedded in their Information Systems for their daily operations. This logic implements the business rules in place in the organization, which must be continuously adapted in response to market changes. Unfortunately, this evolution implies understanding and evolving also the underlying software components enforcing those rules. This is challenging because, first, the code implementing the rules is scattered throughout the whole system and, second, most of the time documentation is poor and out-of-date. This is specially true for older systems that have been maintained and evolved for several years (even decades). In those systems, it is not even clear which business rules are enforced nor whether rules are still consistent with the current organizational policies.**

In this sense, the goal of this paper is to facilitate the comprehension of legacy systems (in particular COBOL-based ones) by providing a model driven reverse engineering framework able to extract and visualize the business logic embedded in them.

I. Introduction

Organizations rely on the logic embedded in their software systems for their daily operations. In a continuously changing environment (new laws and reglamentations, new competitors,...) this logic must evolve quickly in order to align the policies of the organization with the market needs.

A precise understanding of the system, i.e. what business rules[1] the system is currently enforcing, is a prerequisite for its evolution. Unfortunately, this understanding is an error-prone and time-consuming activity because of two main reasons. Firstly, the business logic is usually scattered throughout the code and intertwined with presentation and other technical and auxiliary code. Secondly, documentation (textual descriptions, models of the software,...) is generally not available or it is outdated. These limitations complicate the maintenance and evolution of the system, slowing down the company response to the new requirements constantly demanded by the market and causing, in some cases, inconsistencies between the current organizational policies and the business logic enforced by the system. This is specially true in the case of legacy systems where the original developers may have left the organization. One particular important type of legacy systems are COBOL (COmmon Business Oriented Language) systems which still play a critical role in the business world. They are the focus of this work.

In order to facilitate the comprehension and the evolution of COBOL-based systems, we propose a semi-automatic Business Rule Extraction (BREX) method, that aims at extracting the business logic hard-coded in a system as a set of business rules. These business rules can then be validated (or updated/reimplemented) by the company's stakeholders.

We describe a new BREX approach for COBOL applications based on the principles of Model Driven Engineering (MDE). Thanks to the MDE techniques, we can work at a higher abstraction level on an homogeneous (model-based) representation of the system, which also allows us to benefit from the plethora of available MDE tools for model manipulation, visualization and transformation.

The framework has been created in response to (and in cooperation with) the needs of IBM Rational Software Group to improve the reverse engineering services and tools they offer to their customers. The implementation of the framework has been integrated with IBM Rational Programming Patterns and an early validation with IBM engineers has been performed. In the future, IBM envisages to use the framework to assist its customers on the comprehension of their systems.

This paper is structured as follows: Section 2 provides some basic concepts on COBOL; Section 3 introduces a running example; Section 4-7 describe the framework and its main components; Section 8 depicts the industrial experience; Section 9 discusses the related work and finally Section 10 closes the paper.

II. COBOL Basic Concepts

COBOL is a procedural language that structures programs in 4 divisions: *identification*, *environment*, *data* and *procedure divisions*. They are used respectively to identify the program, to describe the input-ouput data sources, to declare the data structures and to define procedures to access and modify the program data structures.

We focus our analysis on the *data* and *procedure divisions* in order to identify variables and procedures that handle business information.

In the *data division*, two types of data structures can be defined: *data items* and *group items*. The former are variables that specify primitive data types; while the latter are used to represent more complex data structures.

Data items are defined using *PICTURE* clauses according to 5 code characters (i.e., *9*, *V*, *S*, *X* and *A*), that are repeated to define the size (i.e., number of bytes) of a given *data item*. *9*, *V* and *S* deal with numeric representations; they specify respectively a numeric value, the decimal point and the sign of the numeric value. On the other hand, *X* and *A* represent in turn alphanumeric and alphabetic (i.e., A-Z, space) values. These primitive types are specified in *data items*, that can be used to composed *group items*.

Finally, both *data* and *group items* are defined in combination with a level number, that represents the data hierarchy. In particular, 01-49 are reserved for *group* or *data items*; 66 for *renames* clause, that allows regrouping *data items* in a *group item*; 77 for independent *data items*; and finally 88 for condition names, where each of them represents a value of a given conditional variable. The data structures defined in the *data division* are accessed/modified in the *procedure division*.

The *procedure division* is composed by sections, paragraphs, sentences and statements. A section contains paragraphs, a paragraph sentences and a sentence statements.

Special commands on paragraphs, sentences and statements can be used to alter the sequential control flow of a COBOL program. We focus on a sub-set of them: PERFORM, GO TO and NEXT SENTENCE. Iterations on the code are achieved using the PERFORM command, that transfers control to one or more statements and returns control to the next statement after the execution of such statements is completed. If the statements are contained in sequential paragraphs, PERFORM is extended with the word THRU to indicate the first and last executed paragraphs. GO TO statements are used to transfer control from one paragraph to another. NEXT SENTENCE phrases allow assigning control to the first statement of the sentence following that command.

The concepts presented above give a small overview of COBOL and they are needed to understand the following Sections.

III. Running Example

In order to illustrate our framework, a small COBOL program will be used as a running example[1]. The program allows a customer to buy products in a shop, if the latter is open. The shop offers several products, which are represented by an unit price and the available quantity. They can be bought if the customer has enough money and enough room in his bag to put the products in.

The data structures of the program are shown in Fig. 1. The shop is represented as a *group item*, that defines its property (i.e., open/closed variable *OP*) and the unit price and quantity for the products it sells (i.e., vegetables, meat, bread, milk, fruit). The other data structures (i.e., *MONEY*, *REST*, *BAG*, *MAX-CAP*, *NEED*) are *data items* that represent the customer information. In particular, *MONEY* and *REST* are respectively the money owned by the customer (i.e., the initial value is set to 50) and the money left after buying products; *BAG* and *MAX-CAP* are the maximum capacity of the bag and the number of the current products inside; and finally *NEED* defines if a product is needed or not.

[1]The input and output of our framework for the running example can be found at http://docatlanmod.emn.fr/BrexCobolExample/intro.html

```
01 SHOP.
   10 OP          PICTURE 9.
   10 QT-VEG      PICTURE 99.
   10 QT-MEAT     PICTURE 99.
   10 QT-BREAD    PICTURE 99.
   10 QT-MILK     PICTURE 99.
   10 QT-FRUIT    PICTURE 99.
   10 PR-VEG      PICTURE 9.
   10 PR-MEAT     PICTURE 9.
   10 PR-BREAD    PICTURE 9.
   10 PR-MILK     PICTURE 9.
   10 PR-FRUIT    PICTURE 9

   77 MONEY       PICTURE 99, VALUE 50.
   77 REST        PICTURE 99.
   77 BAG         PICTURE 9.
   77 MAX-CAP     PICTURE 9, VALUE 10.
   77 RAND        PICTURE 9.
   77 NEED        PICTURE 9.
```

Fig. 1. Data structures of the running example

The program starts with an initialization paragraph (i.e., *INIT*). If the shop is open, the products with their corresponding quantities and prices are initialized (i.e., *INIT-PRD*). Then, the list of products is scanned by means of five paragraphs: *BUY-VEG*, *BUY-MEAT*, *BUY-BREAD*, *BUY-MILK* and *BUY-FRUIT*. For each of these products, *ISNEEDED* checks whether that product is needed by the customer or not. If it is, the customer can buy it on condition that he has enough money and room in his bag. On the contrary, the program ends printing the information concerning the money left and the number of products bought.

Finally, if the list of the products is entirely browsed, but still enough money and room in the bag are available, a new iteration of that list can be performed.

```
BUY-FRUIT.
PERFORM ISNEEDED THRU ISNEEDED-FN.
a IF NEED = 1 AND QT-FRUIT > 0
    IF MONEY > PR-FRUIT AND BAG < MAX-CAP
      ADD 1 TO BAG
b     COMPUTE MONEY = MONEY - PR-FRUIT
      SUBTRACT 1 FROM QT-FRUIT
    ELSE
      GO TO PRINT
ELSE
   GO TO CHECK.
BUY-FRUIT-FN. EXIT.
```

```
CHECK.
IF MONEY <= 0 OR BAG >= MAX-CAP
   GO TO PRINT
ELSE
   GO TO BUY-VEG.
CHECK-FN. EXIT.
```

```
PRINT.
MOVE MONEY TO REST.
DISPLAY "REST:" MONEY.
DISPLAY "NB OF PRODUCTS:" BAG.
FIN.
```

```
ISNEEDED.
   COMPUTE NEED = FUNCTION RANDOM (1) * 2.
ISNEEDED-FN.EXIT.
```

Fig. 2. BUY-FRUIT and its related paragraphs

The logic embedded in each of the paragraph representing the action of buying a given product follows the same principle. In Fig. 2, the logic coded in *BUY-FRUIT* and its related paragraphs are shown. The variable *NEED* is calculated in the paragraph *ISNEEDED* (for the purposes of the simulation, a random value is assigned to the variable). If the product is needed, there are still units available (*QT-FRUIT* variable), the customer has still some money and enough space in the bag, the product is added to the bag and both customer's money and product's quantity are updated. If one of the previous conditions is not true, paragraphs *CHECK* or *PRINT* are executed triggering the end of program or, depending on

the remaining money and the room left in the bag, a new iteration for buying products.

Despite the simplicity of the running example proposed, it contains different business rules. A manual inspection of the source code allows to identify many of them:

- If the shop is open, then the customer can buy products
- If a product is available, then it may be bought
- If a product P is needed, then the customer buys P
- If the client has enough money, then he can buy products
- If the client has enough room in his bag, he can buy products
- If a product is bought, its quantity is decreased by one
- If a product is bought, its price is decreased from the money of the client

The automatic discovery of such kind of rules is the purpose of this paper.

IV. Framework Description

The proposed MDE-based framework, shown in Fig. 3, consists in 4 phases: an initial *Model Discovery* phase plus the three main steps of a standard BREX process[2], *Variable Identification*, *Business Rule Identification* and *Business Rule Representation*.

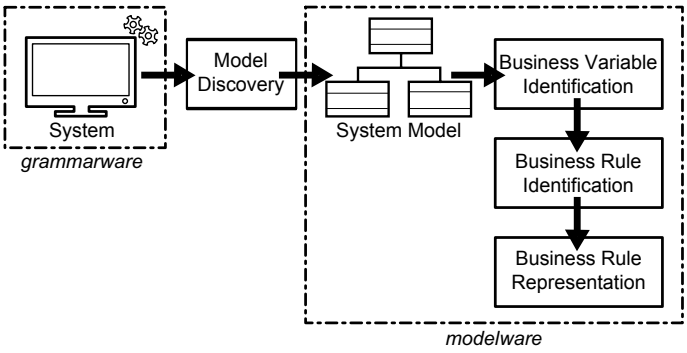

Fig. 3. Framework overview

The additional *Model Discovery* phase is needed to go from the "programming" (or grammar-ware) technical space to the model-ware space[3]. Given a COBOL program, it creates a model-based representation of the source COBOL code. This initial model has a one-to-one correspondence with the code so there is no information loss at this point. Existing tools like MoDisco[4] or IBM COBOL Application Model[2] can be used for this purpose. This model will be then manipulated in the next framework's steps to extract the business rules. In particular, Variable Identification identifies in the code the variables representing business concepts. Business Rule Identification locates business rules using code slicing techniques[5] on the variables found in the previous step. Finally, the Business Rule Representation step visualizes the extracted rules.

Additionally, our framework provides BREX traceability[6], meaning that the framework ties the source code elements to the elements composing the business rules. This helps users navigating back and forth between the rules and the input code. Traceability is implemented by explicitly linking in each phase transition the input model (or code) elements with the corresponding output model elements generated by the model transformations executed in that phase[7].

Next sections explain the Variable Identification, Business Rule Identification and Business Rule Representation steps in detail.

[2]http://tinyurl.com/IBMCobolApplicationModel

V. Variable Identification

The Variable Identification step reduces the number of variables to analyse by filtering out those that are not business relevant. It takes as input the COBOL model and returns the "business" variables.

This step can be manual or automatic. In the first case, the user navigates the code and directly marks the variables to analyse. In the second case, an heuristic-based strategy identifies such variables based on the kind of statements in which they appear (and their role they play).

Firstly, the statements are divided in several groups according to the COBOL command they contain. The groups are: *conditional*, *computation*, *in-out*, *end*, *move*, *goto*, *perform* and *call*. In particular, *conditional* regroups the different kinds of if-statements that exist in COBOL (i.e., if-then, if-then-goto, if-then-else, if-then-else-goto, if-next-else-goto, etc.). *Computation* contains the statements that model mathematical operations (i.e., COMPUTE, ADD, SUBTRACT, etc.). *In-out* group collects the statements used to prompt or get information from input sources (i.e., DISPLAY, ACCEPT). *End* includes commands used to end a program (i.e., STOP RUN, GO BACK, etc.). The remaining groups are composed by only one kind of statements, that is represented by the name of the group.

Secondly, after grouping the statements, for each of them, the variables in it are collected in four categories. *Condition variables* are variables in *if* statement conditions (e.g. NEED and QT-FRUIT at line *a* in Fig. 2); *index variables* are indexes of array structures; *source variables* and *target variables* are respectively the variables affecting and being affected in a statement (e.g. target variable: MONEY, source variables: MONEY and PR-FRUIT at line *b* in Fig. 2).

Based on this classification, it is possible to define different heuristics to identify business variables. According to our COBOL work experience, we have came up with three complementary heuristics such that strong candidates to be classified as business variables are:

1) all target variables in *computation* statements
2) all the variables that appear in *in-out* statements
3) all the variables in *conditional* statements

Note that an hybrid approach is also possible where an automatic step returns a set of candidate variables and then the user filters some of them.

Figure 4 shows the result of the heuristics previously described concerning the running example. All target variables in *computation* statements are depicted on the upper row; while

on the center and on the bottom the variables respectively in *in-out* and *conditional* statements are listed.

Computation	PR-FRUIT, PR-BREAD, QT-VEG, QT-MILK, BAG, QT-BREAD, PR-VEG, NEED, MONEY, PR-MEAT, QT-MEAT,PR-MILK,QT-FRUIT
In-out	MONEY, BAG
Conditional	PR-FRUIT, PR-BREAD, QT-VEG, QT-MILK, BAG, QT-BREAD, PR-VEG, NEED, MONEY, PR-MEAT, QT-MEAT, PR-MILK, QT-FRUIT, MAX-CAP, OP

Fig. 4. Variable identification step for the running example

VI. Business Rule Identification

Business Rule Identification (Fig. 5) discovers the business rules related to the variables obtained in the Variable Identification step by static slicing techniques on the source code. It is composed by three sub-steps. Control Flow Analysis, Data Flow Analysis and Rule Discovery. The global inputs are the model generated from a COBOL program and one or more variables identified in the previous step [3]. The output is a Control Flow Graph (CFG) enriched with the information about the statements and the variables composing the business rules for the given input variable(s).

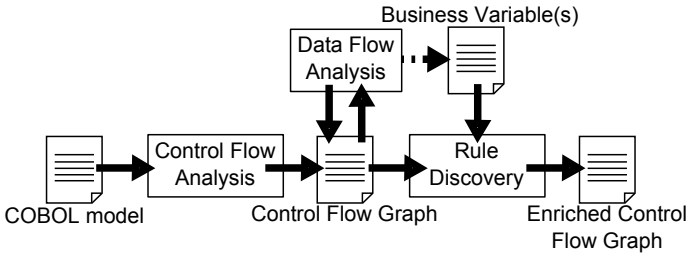

Fig. 5. Business Rule Identification step

A. Control Flow Analysis

This step generates a CFG model from a given COBOL model. The CFG model is derived from the original COBOL model. The generation process for the CFG analyses the syntactic order of the COBOL commands in the initial model and infers the possible execution flows of the program.

The CFG model conforms to the metamodel shown on the center of Fig. 6. The entity *Model* stores *Paragraphs*, *Sentences* and *Statements* composing the program. They are all linked to the entity *Trace*, that is used to support the MDE traceability in the framework (i.e., the attribute *link* stores the references to the corresponding COBOL model entities).

The key element that stores the relationships between the statements is the association *next*. It indicates all possible statements to be executed next after a statement *X* (which one will be next may change on each execution depending on the run-time conditions, so this association collects all possible alternatives). From this *next* association we derive the other

[3]For the sake of comprehension, we describe the process assuming a single variable as input

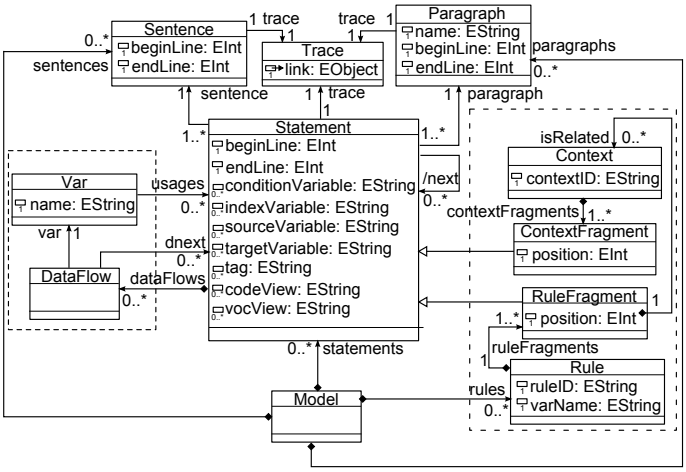

Fig. 6. Data Flow metamodel (left) Control Flow metamodel (center) and Business Rule entities (right)

next-like associations (i.e., *nextSequential*, *nextTransparentIf*, *nextTransparentPerform*, that are not shown in the metamodel due to lack of space) to facilitate the analysis in the following steps of the framework.

The rules to compute the *next* statements of a statement *stat* are listed in Tab. I . The first six rules are related to the type of *stat*; while the following two concern its position. The last rule is applied to all statements not included in the previous rules.

TABLE I
Next statement rule

Current Stat	Next
Type: PERFORM-THRU	first statement of FROM paragraph in PERFORM-THRU
Type: GO-TO	first statement of the GO-TO paragraph
Type: EXIT PROGRAM, STOP RUN, GO BACK	none
Type: IF-THEN-ELSE	first statements in THEN and ELSE branches
Type: NEXT SENTENCE	first statement of the following sentence
Type: IF-THEN	first statements in THEN and after IF stat.
Position: last stat in THEN branch	first statement after IF
Position: last stat in PERFORM-THRU	statement after PERFORM-THRU
Position: - **Type:** -	following statement

The information concerning the variables referenced in statements are stored in the attributes *conditionVariable*, *indexVariable*, *sourceVariable* and *targetVariable*; while the type of the statement is stored in the attribute *tag* (for the sake of simplicity, these concepts are directly represented as Strings instead of appearing as separate classes in Fig. 6). Other attributes store the position of the statement (i.e., begin and end lines) and the *Sentence* and *Paragraph* containing it for traceability purposes.

The remaining entities in the metamodel (i.e., *DataFlow*, *Var*, *Rule*, *RuleFragment*, *Context* and *ContextFragment*) are discussed in the Data Flow Analysis and Rule Discovery steps.

B. Data Flow Analysis

This step is used to find relations among (business-relevant) variables. The information collected can be used to run a Rule Discovery step on a set of related variables. Data Flow Analysis is an optional operation that takes as input the model that represents the CFG of the program and returns the same model enriched with data flow information concerning the variables within that program.

DataFlow and *Var* entities of the CFG metamodel (i.e., on the left in Fig. 6) store the data flow information.

For each variable *v* within a statement *stat*, a *DataFlow* instance is created and it is linked to *stat* by the reference *dataFlows*. A *DataFlow* is defined respectively by a reference to the variable *v* it contains (i.e., *var* in Fig.6) and by a list (i.e., *dnext*) of statements. These are the statements that follow *stat* in the CFG and that contain the same variable. Finally the entity *Var* represents the variable *v*. It contains the name of *v* and a list of statements (i.e., *usages* in Fig.6) where *v* is used.

C. Rule Discovery

Rule discovery relies on program slicing techniques to recover the business rules associated to one or more variables.

A rule represents a possible execution path in the program relevant to a business variable. It includes one or more statements modifying/accessing such variable. According to the possible execution paths, several rules may exist for the same variable. Each rule represents an independent execution path in the code, that is no fully-contained in other ones.

A rule is composed by rule fragments, that are selected statements in the code. A rule fragment can be either a statement *S* where the input variable is referenced or a conditional statement that contains in one of its branches *S*. Optionally, a rule fragment may be associated to contexts. A context contains the remaining conditions in the control flow that trigger that rule fragment. Thus, a context is composed by context fragments, that are the conditions of conditional statements.

Rule Discovery locates the business rules related to a business variable in the program. The inputs of this step are the CFG model and a variable. The output is the CFG enriched with information about the business rules related to that variable. It is divided into two steps: Rule Fragment Identification and Rule Context Identification.

1) Rule Fragment Identification: this process is composed by three phases. Initially, the statements containing the business variable passed as input are located in the CFG. In the second phase, the execution paths including these statements are calculated (i.e. during this calculation, only the statements identified in the first step are added to the execution paths). For each of these execution paths a *Rule* (Fig. 6) is created. It is defined by an identifier *RuleID* and the name of the variable passed as input (*VariableName*).

In the last phase, the conditional statements that include the statements identified in the first step are added to the corresponding paths. Finally, all these statements are stored as *RuleFragments* (Fig. 6) and their locations in the execution path are saved in the attribute *Position*.

```
INIT.
  IF OP = 1
     DISPLAY "SHOP IS OPEN"
     PERFORM INIT-PRD THRU INIT-PRD-FN
     GO TO INIT-FN
  ELSE
     DISPLAY "SHOP IS CLOSED"
     GO TO INIT.
INIT-FN.EXIT.
BUY-VEG.
PERFORM ISNEEDED THRU ISNEED-FN.
IF NEED = 1 AND QT-VEG > 0
   IF MONEY > PR-VEG AND BAG < MAX-CAP
      ADD 1 TO BAG
   ...
   ELSE ...
```

Fig. 7. Example of Rule Fragment Identification

In Fig. 7, the identification of the rule fragments concerning the variable *BAG* is shown. For the sake of comprehension, we focus only on the execution path containing the statement in the paragraph *BUY-VEG*. The selection of the statement *ADD 1 TO BAG* is the result of the two first steps of the Rule Fragment Identification process. In the third step, the conditional statements that include this statement (i.e., *IF NEED = 1 AND QT-VEG* $>$ *0* and *IF MONEY* $>$ *PR-VEG AND BAG* $<$ *MAX-CAP*) are added to the rule.

2) Rule Context Identification: the process to identify the contexts that are related to each *RuleFragment* of a *Rule* is composed by two phases. Firstly, for each *Rule*, the corresponding *RuleFragments* are retrieved from the CFG. Later, each *RuleFragment* is used as backwards starting point to discover the ordered sets of control flow condition that might have been crossed in the program, without passing by other *RuleFragments* of the same *Rule*. Each set of if-conditions represents a *Context* (Fig. 6) and it is defined by an identifier *ContextID*. Any condition in a *Context* set is a *ContextFragment* (Fig. 6) and its location inside the context is stored in the attribute *Position*.

```
INIT.
  IF OP = 1
     DISPLAY "SHOP IS OPEN"
     PERFORM INIT-PRD THRU INIT-PRD-FN
     GO TO INIT-FN
  ELSE
     DISPLAY "SHOP IS CLOSED"
     GO TO INIT.
INIT-FN.EXIT.
BUY-VEG.
PERFORM ISNEEDED THRU ISNEED-FN.
IF NEED = 1 AND QT-VEG > 0
   IF MONEY > PR-VEG AND BAG < MAX-CAP
      ADD 1 TO BAG
   ...
   ELSE ...
```

Fig. 8. Example of Rule Context Identification

In Fig. 8, the paragraph *BUY-VEG* follows the paragraph *INIT*, which is the first in the program. The *RuleFragment*, in

the box on the right, contains the *rule fragments* concerning the variable *BAG*. On the left column, the box contains the *Context*, that is composed by one *ContextFragment* (i.e., *IF OP = 1*), since only this if-condition is crossed to reach the *RuleFragment*.

VII. Business Rule Representation

Business Rule Representation (Fig. 9) is the last step of the framework. Its goal is to generate comprehensible textual and graphical representations of the discovered business rules and their orchestration (i.e. connections and precedences among the rules).

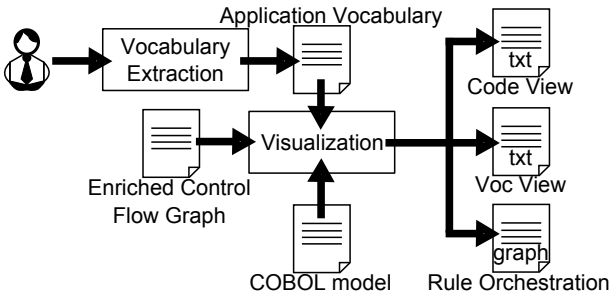

Fig. 9. Business Rule Representation step

This step is composed by two operations: Vocabulary Extraction and Visualization.

A. Vocabulary Extraction

Vocabulary extraction is an optional step simply aimed at providing the set of labels for each variable (in natural language) defined by the user. Figure 10 shows a vocabulary excerpt for the running example. This can be a manual operation or an assisted one.

```
<vocabulary:Model ...>
...
<entries key="OP" value="OPEN" />
<entries key="QT-MEAT" value="QUANTITY MEAT" />
<entries key="PR-MEAT" value="PRICE MEAT" />
<entries key="MAX-CAP" value="MAXIMUM CAPACITY" />
...
</vocabulary:Model>
```

Fig. 10. Running example vocabulary

The vocabulary model conforms to the metamodel presented in Fig. 11. The root element of this metamodel is the entity *Model* that contains a list of *programs*. A *Program* is defined by a *name* and a *label* containing its description. It can have zero or more *entries*. Each *Entry* stores the name of a variable in the program in the attribute *key* and the corresponding verbalization in the attribute *value*.

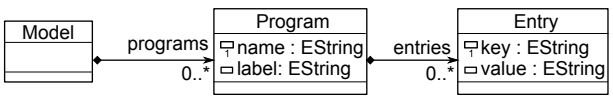

Fig. 11. Vocabulary metamodel

B. Visualization

This step provides artifacts that ease the comprehension of the business rules and their relations. Its inputs are the COBOL model, the corresponding CFG containing the business rule information and optionally the vocabulary of the application. The outputs are text and graph of the gathered rules according to the information contained in the CFG model (Fig.6).

In the following we describe textual and graphical visualizations of the identified business rules.

1) Textual Visualization: all the *Rules* in the CFG are collected to generate the textual representations. The *RuleFragments* composing a *Rule* are retrieved and ordered (thanks to the *Position* attribute) according to their relative positions in the corresponding execution path. For each *RuleFragment*, the related entity in the COBOL model is retrieved and a code textual representation is calculated from it. If the vocabulary has been defined, also a vocabulary-based representation of the *RuleFragment* is created. In this case, the textual representation is calculated mixing the hard-coded translations of the COBOL commands (e.g., the operator $<$ is translated into LESS-THAN, etc.) with the descriptions of the variables in the vocabulary.

Finally these textual representations are stored back in the CFG (i.e. attributes *codeView* and *vocView* of the *Statement* class that is the super-classes of the corresponding *RuleFragment*) and the rules are saved in textual files. Each file will contain separately all the rules discovered for a given variable. The same process is done for the textual representations of *Contexts* and *ContextFragments*.

The example, in Fig. 12, shows the rule *PR-MEAT/PRICE MEAT* for technical and business users, and due to space limitations only the *ContextFragments* of the first *RuleFragment*.

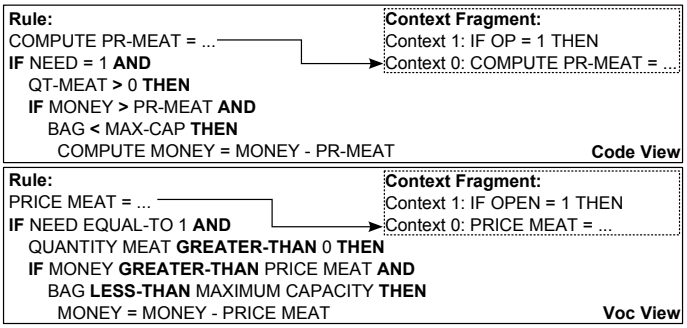

Fig. 12. Example of textual outputs for the rule PR-MEAT/PRICE MEAT

2) Graphical Visualization: relationships between the rules are better displayed by means of a graph-based representation. Orchestration is achieved connecting together the rules that share at least a *RuleFragment*/*Statement*.

In Fig. 13 the three rules concerning the variables *PR-MEAT* (in the box on the right), *PR-BREAD* (in the box on the left) and *MONEY* (in the center) are shown. In this example, the BREX process locates only one rule for each variable. The rules concerning the variables *PR-BREAD* and *PR-MEAT* are connected to the rule related to the variable *MONEY*, since

they share with it a *RuleFragment*. The rule *MONEY* is not shown entirely due to space limitations.

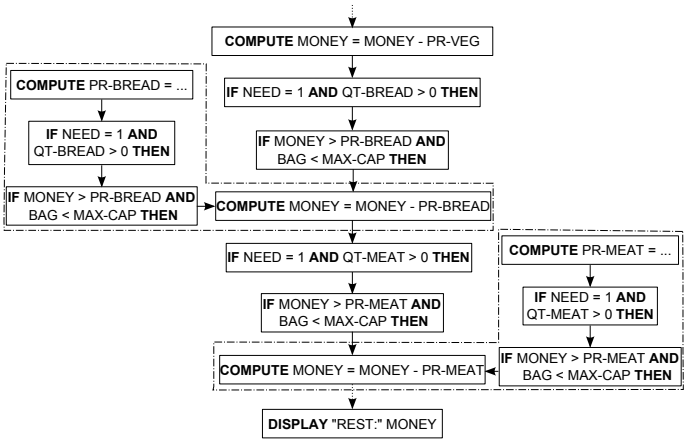

Fig. 13. Orchestration of the rules *PR-MEAT*, *PR-BREAD* and *MONEY*

VIII. Industrial Experience

The framework has been developed within IBM France, that has provided expertise, tools support and a use case. The preliminary validation of the framework has been positive, highlighting the maturity of MDE technologies and the practical feasibility of model-based BREX approaches. In the short-term, IBM plans to test this framework on a complex customer system.

A. Implementation

The framework[8] has been implemented to work with both generic and IBM-specific contexts.

IBM tools have been used throughout the framework. Model Discovery phase relies on COBOL Application Model of IBM Rational Developer[4] (RDZ). RDZ allows to automatically generate a low-level model (basically a model that has a one-to-one correspondence with the COBOL syntax) out of a COBOL program. Variable Identification uses the functionalities provided by IBM Rational Programming Patterns[5] (RPP). RPP is strongly designed on MDE principles, which eases the integration with the framework. RPP allows to attach to any data structure (programs, variables, ...) a label containing a short explanation and provides an interface that facilitates the navigation of all the data structures composing a COBOL system. In this way, it is possible to collect automatically those labels as part of the reverse engineering process and associate them to the corresponding entity. This vocabulary can then be used to improve the visualization of the extracted rules.

Other auxiliary tools are Portolan[9], a model-driven cartography tool used to represent graphically the extracted rules, and the ATL Transformation Language (ATL) [10], used to implement all model manipulation operations required by the three steps of the framework. Both tools are open source.

[4]http://publib.boulder.ibm.com/infocenter/ieduasst/rtnv1r0/index.jsp
[5]http://publib.boulder.ibm.com/infocenter/rppzhelp/v8r0/index.jsp

MDE has facilitated the interoperability among the different tools employed in the framework. The framework is packaged and distributed to its users as an Eclipse plug-in.

B. Early Validation

In order to check the accuracy of the framework, a preliminary experiment has been conducted on a use case provided by IBM. The test has concerned the analysis of an IBM RPP application managing flight and pilots containing 14 programs and 130 variables in around 6500 lines of code.

We asked four internal IBM COBOL experts to analyse the business rules generated by our BREX framework and assess whether the rules were meaningful (i.e. they were actual business rules) and understandable. They had access to the original COBOL code and were given two hours to perform the validation. For the sake of simplicity, instead of generating all rules for the system, we focused on the rules related to a small subset of business variables previously identified.

At the end of validation, they all agreed the framework was able to generate the complete set of rules for the input variables and considered the result useful to understand the COBOL code. The only concern was that the rules were not completely "clean" meaning that some of them still included technical statements (e.g., conditions to check end of the file on read access file operations). This direct use of implementation technology can be removed by improving the Business Rule Representation step to reexpress relevant technical statements in more business-like terms.

IX. Related Work

The discovery of business rules out of legacy source code is a research domain that has been explored extensively though only some of the works focus on COBOL applications. For instance, [11] and [12] propose BREX frameworks respectively for C/C++ and Java; despite our framework shares the same conceptual steps of those approaches, the corresponding heuristics cannot be reused in our context due to the huge differences between COBOL and those languages.

[13] depicts a BREX framework to extract business rules out of COBOL source code. The framework is mainly based on the business rule identification step and the corresponding slicing operation (i.e., backward slicing). On the other hand, it does not provide heuristics concerning the variable identification step. In addition, the business rule presentation is based only on the source code view.

Our work goes beyond, since we provide heuristics for the variable identification step and a higher abstraction level representation of the extracted rules based on the application vocabulary.

In [14], the authors present a manual approach to extract business logic from source code. In particular, they focus on gathering rules that check that important business conditions have not been violated. The heuristics proposed are based on analysing the code that handle error conditions, relying on the assumption that if an error condition occurs within a program,

the conditions that led to it could potentially be describing a business rule violation.

They authors define the rule discovery for those rules that violate the "system boundaries", skipping the analysis of calculations in the code. In addition, they propose a manual method that may represent a weakness when coping with large and complex systems.

In [15] and [16], the authors propose a framework that extracts business rules from COBOL legacy code. The variable identification is based on locating the variables that appear in calculation statements. The business rule identification consists in retrieving the statements that contain these variables. Finally, the business rule representation in [15] is used to generate graph and source code outputs, providing in addition translations from technical to non-technical terms. In the other work [16], the business rule representation is based on heuristics to relate the application documentation to the extracted business rules.

In both works, the authors focus on the identification of single "business statements" (calculations on relevant variables) within the COBOL code. The statements modifying the same variable are not treated together to discover complete business rules. In our framework, according to the possible execution paths in the program we arrange the statements containing a given variable in order to find complete rules.

[17] proposes a BREX framework for COBOL. It provides heuristics to identify business variables (i.e., system's input and output variables, inputs and outputs of each procedure) and to select a slicing criterion (i.e., start point and end point of a procedure respectively for forward and backward slicing, etc.). Finally, three kind of presentation formats are described for the business rules extracted: the code view presents rules as code fragments; formulae view shows the business rules as formulae and input-output dependence view provides a way to trace data-flows input and output together.

In this work, the representation of the rules is not adapted to business users (e.g., no graphical visualization or rule dependencies are shown).

[18] focuses on recoverying the business knowledge out of COBOL legacy systems. Discovery of business rules is mentioned as a possible application of the approach though the process to identify and visualize them is not discussed.

With respect to all these previous works, we provide a non-intrusive traceability support to relate the rules back to the source code and; and in addition, we offer a clear separation between the context and the rule fragments composing the extracted rules.

X. Conclusion and Future Work

In this paper we have presented a MDE framework for extracting business rules out of a COBOL application. The COBOL code is analyzed and sliced to extract the relevant statements for a given business concept. Visualization techniques provide a comprehensive output that facilitates the understanding of the business rules embedded in the COBOL application. Traceability links facilitate navigating from the rules to the corresponding excerpts of COBOL code.

The modularity and high abstraction level provided by MDE allows the framework to be easily adapted to different COBOL dialects. In particular, the framework has been implemented to work with *Pacbase*, the COBOL variant used at IBM. For its implementation, we have benefited from IBM development tools available for COBOL environment.

As further work, we would like to complete the preliminary validation and apply the framework on IBM-customer COBOL systems. Besides, we are aware that business rules can be enforced in any layer of a software system (e.g., including front-end validations, database triggers and checks,...) so we envision to complement this framework with additional modules covering other technologies (see our work with Java [12]), maximizing the reuse opportunities and providing auxiliary modules in charge of merging and checking the consistency of the obtained rules.

References

[1] D. Hay, K. A. Healy, and J. Hall, "Defining Business Rules – What Are They Really?" 2000, http://www.businessrulesgroup.org/first_paper/br01c1.htm.

[2] H. M. Sneed and K. Erdös, "Extracting Business Rules from Source Code," in *WPC*, 1996, pp. 240–247.

[3] J. Bézivin and I. Kurtev, "Model-based Technology Integration with the Technical Space Concept," in *Metainformatics Symposium*, 2005.

[4] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: a generic and extensible framework for model driven reverse engineering," in *ACM*, 2010, pp. 173–174.

[5] M. Weiser, "Program slicing," *Software Engineering, IEEE Transactions on*, no. 4, pp. 352–357, 1984.

[6] H. S. Baxter I., "A Standards-Based Approach to Extracting Business Rules," http://tinyurl.com/semdesignsBREX-pdf.

[7] A. Galvão I., Goknil, "Survey of Traceability Approaches in Model-Driven Engineering," in *EDOC*, 2007, pp. 313–326.

[8] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting Business Rules from COBOL: A Model-Based Tool," in *WCRE*, 2013, to appear.

[9] V. Mahe, S. Martinez Perez, G. Doux, H. Brunelière, and J. Cabot, "PORTOLAN: a Model-Driven Cartography Framework," INRIA, Tech. Rep. RR-7542, 2011.

[10] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.

[11] X. Wang, J. Sun, X. Yang, Z. He, and S. Maddineni, "Business Rules Extraction from Large Legacy Systems," in *CSMR*, 2004, pp. 249–254.

[12] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "A Model Driven Reverse Engineering Framework for Extracting Business Rules Out of a Java Application," in *RuleML*, 2012, pp. 17–31.

[13] H. M. Sneed, "Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment," in *IWPC*, 2001, pp. 167–175.

[14] A. B. Earls, S. M. Embury, and N. H. Turner, "A method for the manual extraction of business rules from legacy source code," *BT Technology Journal*, vol. 20, no. 4, pp. 127–145, 2002.

[15] E. Putrycz and A. W. Kark, "Recovering Business Rules from Legacy Source Code for System Modernization," in *RuleML*, 2007, pp. 107–118.

[16] E. Putrycz and A. W. Kark, "Connecting legacy code, business rules and documentation," in *RuleML*, 2008, pp. 17–30.

[17] H. Huang, "Business Rule Extraction from Legacy Code," in *COMPSAC*, 1996, pp. 162–167.

[18] F. Barbier, G. Deltombe, P. O., and K. Youbi, "Model Driven Reverse Engineering: Increasing Legacy Technology Independence," in *IWRE*, 2011.